SANA-II Network Device Driver Specification

Amiga Networking Group
Brian Jackson, Dale Larson, Greg Miller, and Kenneth Dyke

The SANA-II Network Device Driver Specification is a standard for an Amiga software interface between networking hardware and network protocol stacks (or for software tools such as network monitors). A network protocol stack is a layer of software that network applications use to address particular processes on remote machines and to send data reliably in spite of hardware errors. There are several common network protocol stacks including TCP/IP, OSI, AppleTalk, DECNet and Novell.

SANA-II device drivers are intended to allow multiple network protocol stacks running on the same machine to share one network device. For example, the TCP/IP and AppleTalk protocol stacks could both run on the same machine over one ethernet board. The device drivers are also intended to allow network protocol stacks to be written in a hardware-independent fashion so that a different version of each protocol stack doesn't have to be written for each networking hardware device.

The standard does not address the writing of network applications. Application writers must not use SANA-II Device Drivers directly. Network applications must use the API provided by the network protocol software the application supports. There is not an Amiga standard network API at the time of this writing, though there is the AS225 TCP/IP package and its *socket.library* as well as other (third-party) packages.

To write a SANA-II device driver, you will need to be familiar with the specification documents for the hardware you are writing to and with the SANA-II Network Device Driver Specification.

To write a network protocol stack which will use SANA-II device drivers, you should have general familiarity with common network hardware and must be very familiar with the SANA-II Network Device Driver Specification as well as the specification for the protocol you are developing. If you are creating a new protocol, you must obtain a protocol type number for any hardware on which your protocol will be used.

Commodore supports the SANA-II specification by providing drivers for the Commodore-Amiga network hardware. We have an *A2065.device* (Ethernet) and intend to produce an *A2060.device* (ARCNET). We also try to examine review copies of third-party SANA-II networking hardware and software to try to make sure that they interoperate with our products.

This standard has undergone several drafts with long periods for comment from developers and the Amiga community at large. These drafts include a UseNet release which was also distributed on the Fish Disks in June, 1991 (as well as published in the '91 DevCon notes), and the November 7 Draft for Final Comment and Approval distributed via Bix, ADSP and UseNet. There were also several intermediate drafts with more limited distribution.

This version of the specification is final. Any new version of the standard (i.e., to add new features) is planned to be backward compatible. No SANA-II device driver or software utilizing those drivers should be written to any earlier version of the specification.

Distribution of this version of the standard is unlimited. Anyone may write Amiga software which implements a SANA-II network device driver or which calls a SANA-II network device driver without restriction and may freely distribute such software that they have written. Amiga is a registered trademark of Commodore-Amiga, Inc. Ethernet is a trademark of Xerox Corporation. ARCNET is a trademark of Datapoint Corporation. DECNet is a trademark of Digital Equipment Corporation. AppleTalk is a trademark of Apple Computer, Inc.

It is important to try to test each SANA-II device driver against all software which uses SANA-II devices. Available example programs are valuable in initial testing. The Amiga Networking Group is interested in receiving evaluation and/or beta test copies of all Amiga networking hardware, SANA-II device drivers and software which uses SANA-II devices. However, we make no assurances regarding any testing which we may or may not perform with such evaluation copies. Contact:

Amiga Networking Group
Commodore International Services Corporation
Technology Group
1200 Wilson Drive
West Chester, PA 19380, USA

Driver Form

SANA-II device drivers are Amiga Exec device drivers. They use an extended IORequest structure and a number of extended commands for tallying network statistics, sending broadcasts and multicasts, network addressing and the handling of unexpected packets. The *Amiga ROM Kernel Reference Manual: Devices* includes information on how to construct an Exec device.

Opening a SANA-II Device

As when opening any other Exec device, on the call to `OpenDevice()` a SANA-II device receives an IORequest structure which the device initializes for the opener's use. The opener must copy this structure if it desires to use multiple asynchronous requests. The SANA-II IORequest is defined as follows:

```
struct IOSana2Req
{
    struct IORequest ios2_Req;
    ULONG ios2_WireError;
    ULONG ios2_PacketType;
    UBYTE ios2_SrcAddr[SANA2_MAX_ADDR_BYTES];
    UBYTE ios2_DstAddr[SANA2_MAX_ADDR_BYTES];
    ULONG ios2_DataLength;
    APTR *ios2_Data;
    APTR *ios2_StatData;
    APTR *ios2_BufferManagement;
};
```

ios2_Req - A standard Exec device IORequest.

ios2_WireError - A more specific device code which may be set when there is an io_Error. See *<devices/sana2.h>* for the defined WireErrors.

ios2_PacketType - The type of packet requested. See the section on ‘‘Packet Types’’.

ios2_SrcAddr - The device fills in this field with the interface (network hardware) address of the source of the packet that satisfied a read command. The bytes used to hold the address will be left justified but the bit layout is dependent on the particular type of network.

ios2_DstAddr - Before the device user sends a packet, it fills this with the interface destination address of the packet. On receives, the device fills this with the interface destination address. Other commands may use this field differently (see the SANA-II Network Device Driver Autodocs). The bytes used to hold the address will be left justified but the bit layout is dependent on the particular type of network.

ios2_DataLength - The device user initializes this field with the amount of data available in the Data buffer before passing the IOSana2Req to the device. The device fills in this field with the size of the packet data as it was sent on the wire. This does not include the header and trailer information. Depending on the network type and protocol type, the driver may have to calculate this value. This is generally used only for reads and writes (including broadcast and multicast).

ios2_Data - A pointer to some abstract data structure containing packet data. Drivers may not directly manipulate or examine anything pointed to by Data! This is generally used only for reads and writes (including broadcast and multicast).

ios2_StatData - Pointer to a structure in which to place a snapshot of device statistics. The data area must be long word aligned. This is only used on calls to the statistics commands.

ios2_BufferManagement - The opener places a pointer to a tag list in this field before calling `OpenDevice()`. Functions pointed to in the tag list are called by the device when processing IORequests from the opener. When returned from `OpenDevice()`, this field contains a pointer to driver-private information used to access these functions. See ‘‘Buffer Management’’ below for more details.

The flags used with the device on `OpenDevice()` are (`SANA2OPB_xxx`):

`SANA2OPB_MINE` - Exclusive access to the unit requested.
`SANA2OPB_PROM` - Promiscuous mode requested. Hardware which supports promiscuous mode allows all packets sent over the wire to be captured whether or not they are addressed to this node.

The flags used during I/O requests are (`SANA2IOB_xxx`):

`SANA2IOB_RAW` - Raw packet read/write requested. Raw packets should include the entire data-link layer packet. Devices with the same hardware device number should have the same raw packet format.
`SANA2IOB_BCAST` - Broadcast packet (received).
`SANA2IOB_MCAST` - Multicast packet (received).
`SANA2IOB_QUICK` - Quick IO requested.

Buffer Management

Unlike most other Exec Device drivers, SANA-II drivers have no internal buffers. Instead, they read/write to/from an abstract data structure allocated by the driver user. The driver accesses these buffers only via functions that the driver user provides to the driver. The driver user must provide two functions--one copies data to the abstract data structure and one copies data from the abstract data structure. The driver user can therefore choose the data structure used for buffer management by both the driver and driver user in order to have efficient memory and CPU usage overall.

The IOSana2Req contains a pointer to data and the length of said data. A driver is not allowed to make assumptions about how the data is stored. The driver cannot directly manipulate or examine the buffer in any manner. The driver can only access the buffer by calling the functions provided by the driver user.

Before calling `OpenDevice()`, the driver user points ios2_BufferManagement to a list of tags (defined in *<devices/sana2.h>*) which include pointers to the buffer management functions required by the driver (defined below). The driver will fail to open if the driver user does not supply all of the required functions. If the device opens successfully, the driver sets ios2_BufferManagement to a value which this opener must use in all future calls to the driver. This ‘‘magic cookie’’ is used from then on to access these functions (a ‘‘magic cookie’’ is a value which one software entity passes to another but which is only meaningful to one of the software entities). The driver user may not use the ‘‘magic cookie’’ in any way--it is for the driver to do with as it wishes. The driver could in theory choose to just copy the tag list to driver-owned memory and then parse the list for every IORequest, but it is much more efficient for the driver to create some sort of table of functions and to point ios2_BufferManagement to that table.

The specification currently includes only two tags for the `OpenDevice()` ios2_BufferManagement tag list:

`S2_CopyToBuff` - This is a pointer to a function which conforms to the CopyToBuff Autodoc.
`S2_CopyFromBuff` - This is a pointer to a function which conforms to the CopyFromBuff Autodoc.

Packet Type

Network frames always have a type field associated with them. These type fields vary in length, position and meaning by frame type (frame types generally correspond one-to-one with hardware types, but see ‘‘Ethernet Packet Types’’ below). The meanings of the type numbers are always carefully defined and every type number is registered with some official body. Do not use a type number which is not registered for any standard hardware you use or in a manner inconsistent with that registration.

The type field allows the SANA-II device driver to fulfill `CMD_READs` based on the type of packet the driver user wants. Multiple protocols can therefore run over the same wire using the same driver without stepping on each other’s toes.

Packet types are specified as a long word. Unfortunately, the type field means different things on different wires. Driver users must allow their software to be configured with a SANA-II device name, unit number and the type number(s) used by the protocol stack with each device. This way, if new hardware becomes available, a hardware manufacturer can supply a listing of type assignments to configure pre-existing software.

Ethernet Packet Types

Ethernet has a special problem with packet types. Two types of ethernet frames can be sent over the same wire--ethernet and 802.3. These frames differ in that the Type field of an ethernet frame is the Length field of an 802.3 frame. This creates a problem in that demultiplexing incoming packets can be cumbersome and inefficient, as well as requiring driver users to be aware of the frame type used.

All 802.3 frames have numbers less than 1500 in the Type field. The only frames with numbers less than 1500 in the type field are 802.3 frames. SANA-II ethernet drivers abnormally return packets contained in ethernet frames when the requested Type falls within the 802.3 range--if the Type requested is within the 802.3 range, the driver returns the next packet contained within an 802.3 frame, regardless of the type specified for the packet within the 802.3 frame. This requires that there be no more than one driver user requesting 802.3 packets and that it do its own interpretation of the frames.

ARCNET Frames

ARCNET also has a special problem with framing. ARCNET frames consist of a hardware header and a software header. The software header is in the data area of the hardware packet, and includes at least the protocol ID.

There are two types of software header. Old-style ARCNET software headers consist entirely of a one or two byte protocol ID. New ARCNET software headers (defined in RFC 1201 and in the paper ‘‘ARCNET Packet Header Definition Standard’’, Novell, Inc., 1989) include more information. They allow more efficient use of ARCNET through data link layer fragmentation and reassembly (ARCNET has a small Maximum Transmission Unit) and allow sending any size packet up to the MTU (rather than requiring that packets of size 253, 254 and 255 be padded to at least 256 bytes).

SANA-II device drivers for ARCNET should implement the old ARCNET packet headers. Driver users which wish to interoperate with platforms using the new software headers must add the new fields to the data to be sent and must process it for incoming data. A SANA-II driver which implemented the data link layer fragmentation internally (and advertised a large MTU) could be more efficient than requiring the driver user to do it. This would make driver writing more difficult and reduce interoperability, but if there is ever a demand for that extra performance, a new hardware type may be assigned by Commodore for SANA-II ARCNET device drivers which implement the new framing.

Addressing

In the SANA-II standard, network hardware addresses are stored in an array of n bytes. No meaning is ascribed by the standard to the contents of the array.

In case there exists a network which does not have an address field consisting of a number of bits not divisible by eight, add pad bits at the end of the bit stream. For example, if an address is ten bits long it will be stored like this:

```
98765432 10PPPPPP
BYTE 0   BYTE 1
```

Where the numerals are bit numbers and ’P’ is a pad (ignored) bit.

Driver users which do not implement the bit shifting necessary to use a network with such addressing (if one exists) should at least check the number of significant bits in the address field (returned from the device's S2_DEVICEQUERY function) to make sure that it is evenly divisible by eight.

Driver users will map hardware addresses to protocol addresses in a protocol and hardware dependent manner, as described by the relevant standards (i.e., RFC 826 for TCP/IP over Ethernet, RFC 1201 or 1051 for TCP/IP over ARCNET). Some protocols will always use the same mapping on all hardware, but other protocols will have particular address mapping schemes for some particular hardware and a reasonable default for other (unknown) hardware.

Some SANA-II devices will have ''hardware addresses'' which aren't really hardware addresses. As an example, consider PPP (Point-to-Point Protocol). PPP is a standard for transmitting IP packets over a serial line. It uses IP addresses negotiated during the establishment of a connection. In a SANA-II driver implementation of PPP, the driver would negotiate the address at S2_CONFIGINTERFACE. Thus, the address in SrcAddr returned by the device on an S2_CONFIGINTERFACE (or in a subsequent S2_GETSTATIONADDRESS) will be a protocol address, not a true hardware address.

Note: Some hardware always uses a ROM hardware address. Other hardware which has a ROM address or is configurable with DIP switches may be overridden by software. Some hardware always dynamically allocates a new hardware address at initialization. See ''Configuration'' for details on how this is handled by driver writers and by driver users.

Hardware Type

The HardwareType returned by the device's S2_DEVICEQUERY function is necessary for those protocols whose standards require different behavior on different hardware. It is also useful for determining appropriate packet type numbers to use with the device. The HardwareType values already issued for standard network hardware are the same as those in RFC 1060 (assigned numbers). Hardware developers implementing networks without a SANA-II hardware number must contact CATS to have a new hardware type number assigned. Driver users should all have reasonable defaults which can be used for hardware with which they are not familiar.

Errors

The SANA-II extended IORequest structure (struct IOSana2Req) includes both the ios2_Error and ios2_WireError fields. Driver users must always check IOSana2Reqs on return for an error in ios2_Error. ios2_Error will be zero if no error occurred, otherwise it will contain a value from *<exec/errors.h>* or *<devices/sana2.h>*. If there was an error, there may be more specific information in ios2_WireError. Drivers are required to fill in the WireError if there is an applicable error code.

Error codes are #defined in the ‘‘defined errors’’ sections of *<devices/sana2.h>*:

IOSana2Req S2io_Error field (S2ERR_xxx):

S2ERR_NO_RESOURCES	- Insufficient resources available.
S2ERR_BAD_ARGUMENT	- Noticeably bad argument.
S2ERR_BAD_STATE	- Command inappropriate for current state.
S2ERR_BAD_ADDRESS	- Noticeably bad address.
S2ERR_MTU_EXCEEDED	- Write data too large.
S2ERR_NOT_SUPPORTED	- Command is not supported by this driver. This is similar to IOERR_NOCMD as defined in *<exec/errors.h>* but S2ERR_NOT_SUPPORTED indicates that the requested command is a valid SANA-II command and that the driver does not support it because the hardware is incapable of supporting it (e.g., S2_MULTICAST). Note that IOERR_NOCMD is still valid for reasons other than a lack of hardware support (i.e., commands which are no-ops in a SANA-II driver).
S2ERR_SOFTWARE	- Software error of some kind.
S2ERR_OUTOFSERVICE	- When a hardware device is taken off-line, any pending requests are returned with this error.

See also the standard errors in *<exec/errors.h>*.

IOSana2Req S2io_WireError field (S2WERR_xxx):

S2WERR_NOT_CONFIGURED	- Command requires unit to be configured.
S2WERR_UNIT_ONLINE	- Command requires that the unit be off-line.
S2WERR_UNIT_OFFLINE	- Command requires that the unit be on-line.
S2WERR_ALREADY_TRACKED	- Protocol is already being tracked.
S2WERR_NOT_TRACKED	- Protocol is not being tracked.
S2WERR_BUFF_ERROR	- Buffer management function returned an error.
S2WERR_SRC_ADDRESS	- Problem with the source address field.
S2WERR_DST_ADDRESS	- Problem with destination address field.
S2WERR_BAD_BROADCAST	- Problem with an attempt to broadcast.
S2WERR_BAD_MULTICAST	- Problem with an attempt to multicast.
S2WERR_MULTICAST_FULL	- Multicast address list full.
S2WERR_BAD_EVENT	- Event specified is unknown.
S2WERR_BAD_STATDATA	- The S2IO_StatData pointer or the data it points to failed a sanity check.
S2WERR_IS_CONFIGURED	- Attempt to reconfigure the unit.
S2WERR_NULL_POINTER	- A NULL pointer was detected in one of the arguments. S2ERR_BAD_ARGUMENT should always be the S2ERR.

Standard Commands

See the SANA-II Network Device Driver Autodocs for full details on each of the SANA-II device commands. Extended commands are explained in the sections below.

Many of the Exec device standard commands are no-ops in SANA-II devices, but this may not always be the case. For example, CMD_RESET might someday be used for dynamically reconfiguring hardware. This should present no compatibility problems for properly written drivers.

Broadcast and Multicast

```
S2_ADDMULTICASTADDRESS          S2_MULTICAST
S2_DELMULTICASTADDRESS          S2_BROADCAST
```

Some hardware supports broadcast and/or multicast. A broadcast is a packet sent to all other machines. A multicast is a packet sent to a set of machines. Drivers for hardware which does not allow broadcast or multicast will return ios2_Error S2ERR_NOT_SUPPORTED as appropriate.

To send a broadcast, use S2_BROADCAST instead of CMD_WRITE. Broadcasts are received just like any other packets (using a CMD_READ for the appropriate packet type).

To send a multicast, use S2_MULTICAST instead of CMD_WRITE. The device keeps a list of addresses that want to receive multicasts. You add a receiver's address to this list by using S2_ADDMULTICASTADDRESS. The receiver then posts a CMD_READ for the type of packet to be received. Some SANA-II devices which support multicast may have a limit on the number of addresses that can simultaneously wait for packets. Always check for an S2WERR_MULTICAST_FULL error return when adding a multicast address.

Note that when the device adds a multicast address, it is usually added for all users of the device, not just the driver user which called S2_ADDMULTICASTADDRESS. In other words, received multicast packets will fill a read request of the appropriate type regardless of whether the requesting driver user is the same one which added the multicast address.

In general, driver users should not care how received packets were sent (normally or broadcast/multicast), only that it was received. If a driver user really must know, however, it can check for SANA2IOB_BCAST and/or SANA2IOB_MCAST in the ios2_Flags field.

Drivers should keep a count for the number of opens on a multicast address so that they don't actually remove it until it has been S2_DELMULTICASTADDRESS'd as many times as it has been S2_ADDMULTICASTADDRESS'd.

Stats

```
S2_TRACKTYPE          S2_GETTYPESTATS          S2_GETGLOBALSTATS
S2_UNTRACKTYPE        S2_GETSPECIALSTATS       S2_READORPHAN
```

There are many statistics which may be very important to someone trying to debug, tune or optimize a protocol stack, as well as to the end user who may need to tune parameters or investigate a problem. Some of these statistics can only be kept by the SANA-II driver, thus there are several required and optional statistics and commands for this purpose.

`S2_TRACKTYPE` tells the device driver to gather statistics for a particular packet type. `S2_UNTRACKTYPE` tells it to stop (keeping statistics by type causes the driver to use additional resources). `S2_GETTYPESTATS` returns any statistics accumulated by the driver for a type being tracked (stats are lost when a type is `S2_UNTRACKTYPE`'d). Drivers are required to implement the functionality of type tracking. The stats are returned in a struct Sana2PacketTypeStats:

```
struct Sana2PacketTypeStats
{
    ULONG PacketsSent;
    ULONG PacketsReceived;
    ULONG BytesSent;
    ULONG BytesReceived;
    ULONG PacketsDropped;
};
```

PacketsSent - Number of packets of a particular type sent.
PacketsReceived - Number of packets of a particular type that satisfied a read command.
BytesSent - Number of bytes of data sent in packets of a particular type.
BytesReceived - Number of bytes of data of a particular packet type that satisfied a read command.
PacketsDropped - Number of packets of a particular type that were received while there were no pending reads of that packet type.

`S2_GETGLOBALSTATS` returns global statistics kept by the driver. Drivers are required to keep all applicable statistics. Since all are applicable to most hardware, most drivers will maintain all statistics. The stats are returned in a struct Sana2DeviceStats:

```
struct Sana2DeviceStats
{
    ULONG PacketsReceived;
    ULONG PacketsSent;
    ULONG BadData;
    ULONG Overruns;
    ULONG UnknownTypesReceived;
    ULONG Reconfigurations;
    struct timeval LastStart;
};
```

PacketsReceived - Number of packets that this unit has received.
PacketsSent - Number of packets that this unit has sent.
BadData - Number of bad packets received (i.e., hardware CRC failed).
Overruns - Number of packets dropped due to insufficient resources available in the network interface.
UnknownTypeReceived - Number of packets received that had no pending read command with the appropriate packet type.
Reconfigurations - Number of network reconfigurations since this unit was last configured.
LastStart - The time when this unit last went on-line.

S2_GETSPECIALSTATS returns any special statistics kept by a particular driver. Each new wire type will have a set of documented, required statistics for that wire type and a standard set of optional statistics for that wire type (optional because they might not be available from all hardware). The data returned by S2_GETSPECIALSTATS will require wire-specific interpretation. See *<devices/sana2specialstats.h>* for currently defined special statistics. The statistics are returned in the following structures:

```
struct Sana2SpecialStatRecord
{
    ULONG Type;
    ULONG Count;
    char *String;
};
```

Type - Statistic identifier.
Count - Statistic itself.
String - An identifying, null-terminated string for the statistic. Should be plain ASCII with no formatting characters.

```
struct Sana2SpecialStatHeader
{
    ULONG RecordCountMax;
    ULONG RecordCountSupplied;
    struct Sana2SpecialStatRecord[RecordCountMax];
};
```

RecordCountMax - There is space for this many records into which statistics may be placed.
RecordCountSupplied - Number of statistic records supplied.

S2_READORPHAN is not, strictly speaking, a statistical function. It is a request to read any packet of a type for which there is no outstanding CMD_READ. S2_READORPHAN might be used in the same manner as many statistics, though, such as to determine what packet types are causing overruns, etc.

Configuration

S2_DEVICEQUERY S2_CONFIGINTERFACE S2_GETSTATIONADDRESS

The device driver needs to configure the hardware before using it. The driver user must know some network hardware parameters (hardware address and MTU, for example) when using it. These commands address those needs.

When a driver user is initialized, it should try to S2_CONFIGINTERFACE even though an interface can only be configured once and someone else may have done it. Before you call S2_CONFIGINTERFACE, first call S2_GETSTATIONADDRESS to determine the factory address (if any). Also provide for user-override of the factory address (that address may be optional and the user may need to override it). When S2_CONFIGINTERFACE returns, check the ios2_SrcAddr for the actual address the hardware has been configured with. This is because some hardware (or serial line standards such as PPP) always dynamically allocates an address at initialization.

Driver users will want to use S2_DEVICEQUERY to determine the MTU and other characteristics of the network. The structure returned from S2_DEVICEQUERY is defined as:

```
struct Sana2DeviceQuery
{
    ULONG SizeAvailable;
    ULONG SizeSupplied;
    ULONG DevQueryFormat;
    ULONG DeviceLevel;
    UWORD AddrFieldSize;
    ULONG MTU;
    ULONG BPS;
    ULONG HardwareType;
};
```

SizeAvailable - Size, in bytes, of the space available in which to place device information. This includes both size fields.
SizeSupplied - Size, in bytes, of the data supplied.
DevQueryFormat - The format defined here is format 0.
DeviceLevel - This spec defines level 0.
AddrFieldSize - The number of bits in an interface address.
MTU - Maximum Transmission Unit, the size, in bytes, of the maximum packet size, not including header and trailer information.
BPS - Best guess at the raw line rate for this network in bits per second.
HardwareType - Specifies the type of network hardware the driver controls.

On-line

S2_ONLINE S2_ONEVENT S2_OFFLINE

In order to run hardware tests on an otherwise live system, the S2_OFFLINE command allows the SANA-II device driver to be ‘‘turned off’’ until the tests are complete and an ONLINE is sent to the driver. S2_ONLINE causes the interface to re-configure and re-initialize. Any packets destined for the hardware while the device is off-line will be lost. All pending and new requests to the driver shall be returned with S2ERR_OUTOFSERVICE when a device is off-line.

All driver users must understand that any IO request may return with S2ERR_OUTOFSERVICE because the driver is off-line (any other program may call S2_OFFLINE to make it so). In such an event, the driver will usually want to wait until the unit comes back on-line (for the program which called S2_OFFLINE to call S2_ONLINE). It may do this by calling S2_ONEVENT to wait for S2EVENT_ONLINE. S2_ONEVENT allows the driver user to wait on various events.

A driver must track events, but may not distinguish between some types of events. Drivers return S2_ONEVENT with S2_ERR_NOT SUPPORTED and S2WERR_BAD_EVENT for unsupported Events. One error may cause more than one Event (see below). Errors which seem to have been caused by a malformed or unusual request should not generally trigger an event.

Event types (S2EVENT_xxx):

ERROR - Return when any error occurs.
TX - Return on any transmit error (always an error).
RX - Return on any receive error (always an error).
ONLINE - Return when unit goes on-line or return immediately if unit is already on-line (not an error).
OFFLINE - Return when unit goes off-line or return immediately if unit is already off-line (not an error).
BUFF - Return on any buffer management function error (always an error).
HARDWARE - Return when any hardware error occurs (always an error, may be a TX or RX, too).
SOFTWARE - Return when any software error occurs (always an error, may be a TX or RX, too).

Acknowledgments

Many people and companies have contributed to the SANA-II Network Device Driver Specification. The original SANA-II Autodocs and includes were put together by Ray Brand, Perry Kivolowitz (ASDG) and Martin Hunt. Those original documents evolved to their current state and grew to include this document at the hands of Dale Larson and Greg Miller. Brian Jackson and John Orr provided valuable editing. Randell Jesup has provided sage advice on several occasions. The buffer

management callback mechanism was his idea. Dale Luck (GfxBase) and Rick Spanbauer (Ameristar Technologies) have provided valuable comments throughout the process. Nicolas Benezan (ADONIS) provided many detailed and useful comments on weaknesses in late drafts of the specification. Thanks to all the above and the numerous others who have contributed with their comments, questions and discussions.

Unresolved Issues

Unfortunately, it isn't possible to completely isolate network protocols from the hardware they run on. Hardware types and addressing both remain somewhat hardware-dependent in spite of our efforts. See the ''Packet Type'' section for an explanation of how packet types are handled and why protocols cannot be isolated from them. See the ''Addressing'' section for an explanation of how addressing is handled any why protocols cannot be isolated from it.

Additionally, there are at least two cases where a hardware type has multiple framing methods in use (ethernet/802.3 and arcnet/(Novell) ''ARCNET Packet Header Definition Standard''). In both cases, software which must interoperate with other platforms on this hardware may need to be aware of the distinctions and may have to do extra processing in order to use the appropriate frame type. See the sections on ''Ethernet Packet Types'' and on ''ARCNET frames'' for more details.